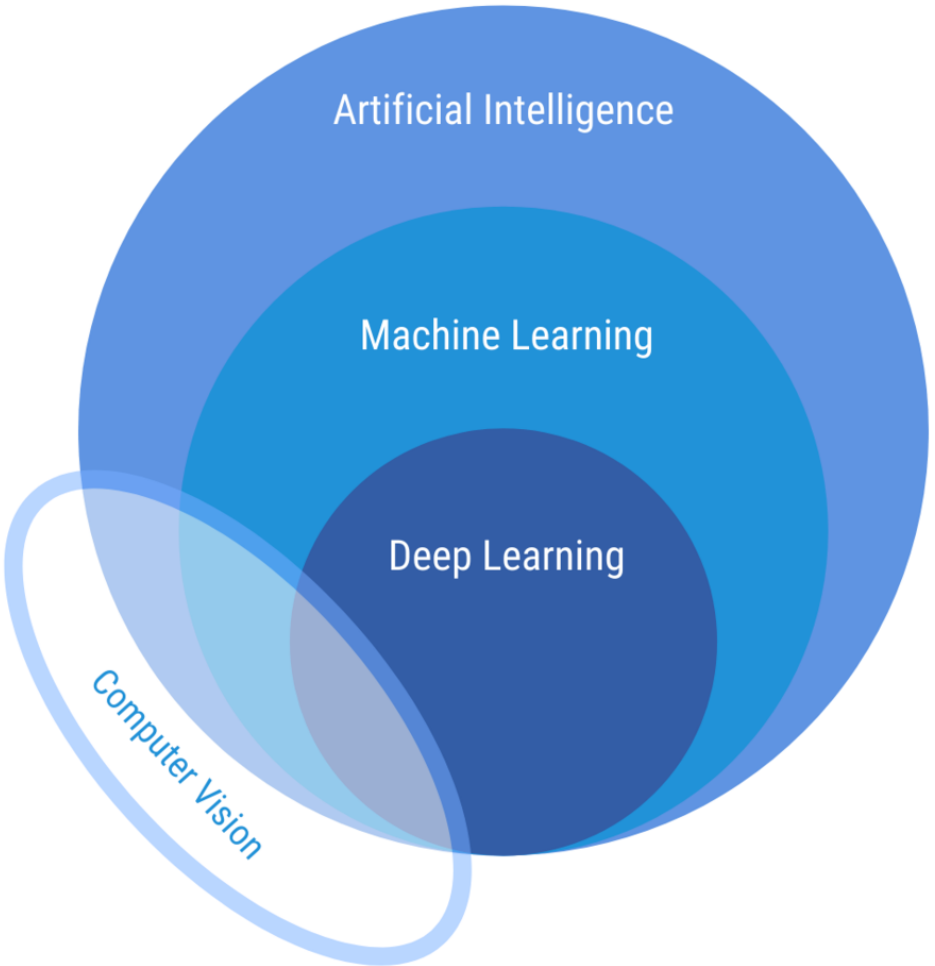
# Computer Vision

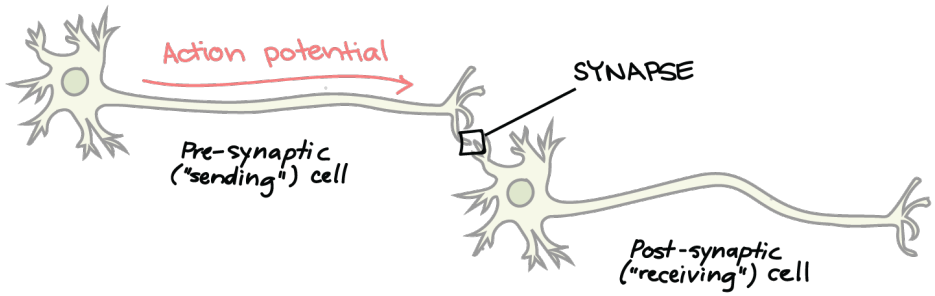
## Deep Learning

Luis F. Teixeira | M.EIC | FEUP


Artificial Intelligence
Machine Learning
Deep Learning
Computer Vision

# Artificial Neural Networks

- Idea: **mimic the brain to do computation**

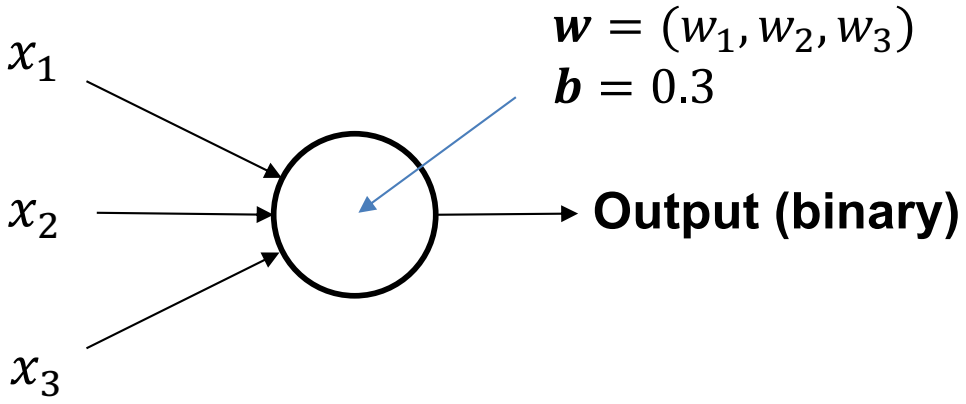



Source: Khan Academy

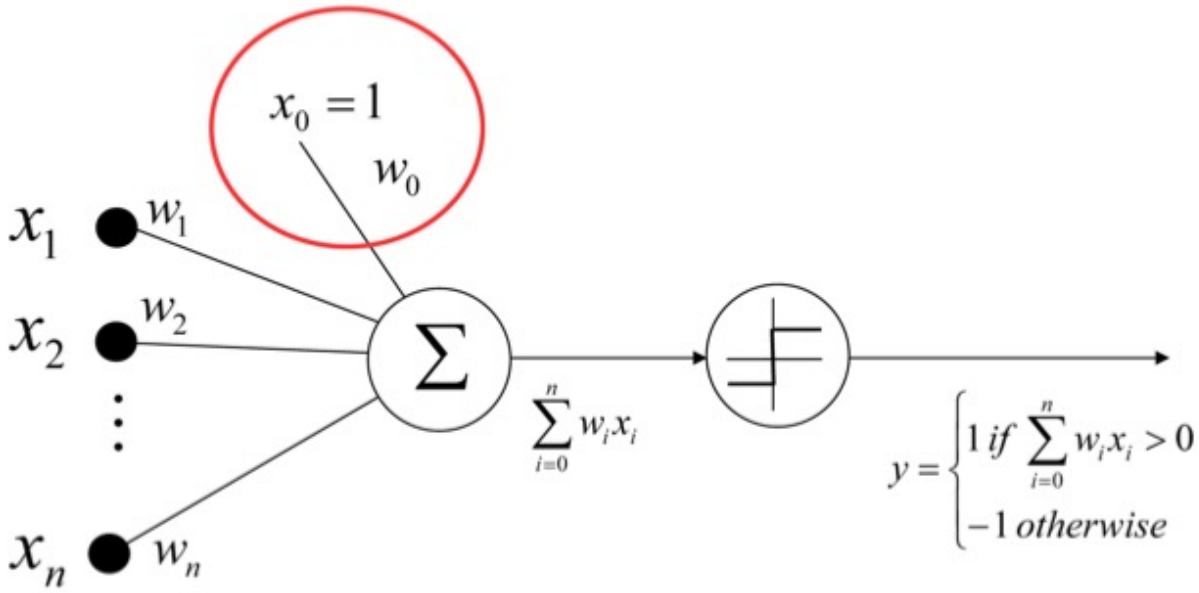
- Artificial neural network:
  - Nodes (a.k.a. units) correspond to neurons
  - Links correspond to synapses
- Computation:
  - Numerical signal transmitted between nodes corresponds to chemical signals between neurons
  - Nodes modifying numerical signal corresponds to neurons firing rate

# Perceptron

- Basic building block for composition is a *perceptron* (Rosenblatt 1958)
- Vector of weights *w* and a 'bias' *b*



# Perceptron – Decision

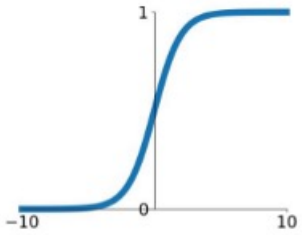



$$y = sign(w^T x)$$

# Activation functions

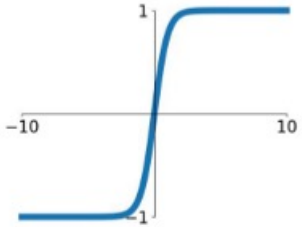
**Sigmoid**

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



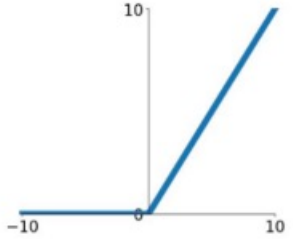

**tanh**

$$\tanh(x)$$



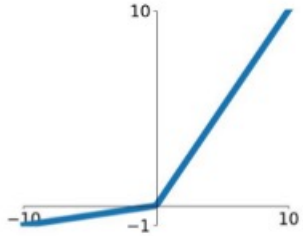

**ReLU**

$$\max(0, x)$$




**Leaky ReLU**

$$\max(0.1x, x)$$




**Maxout**

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

**ELU**

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$

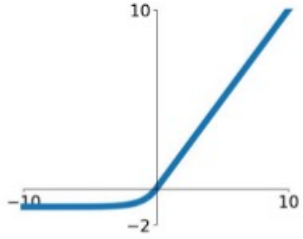
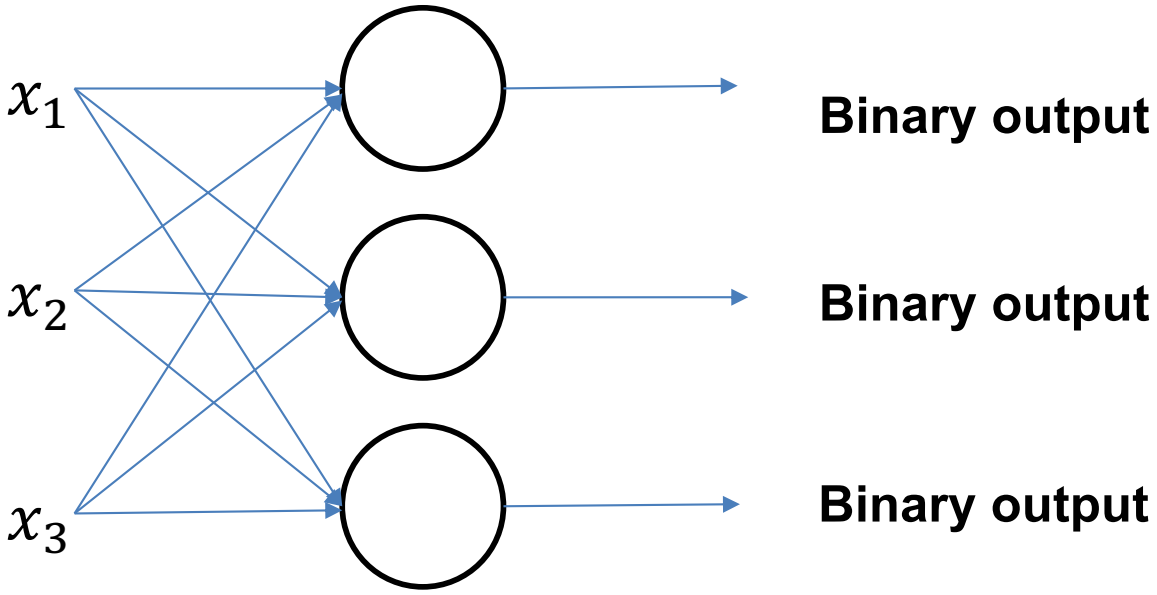


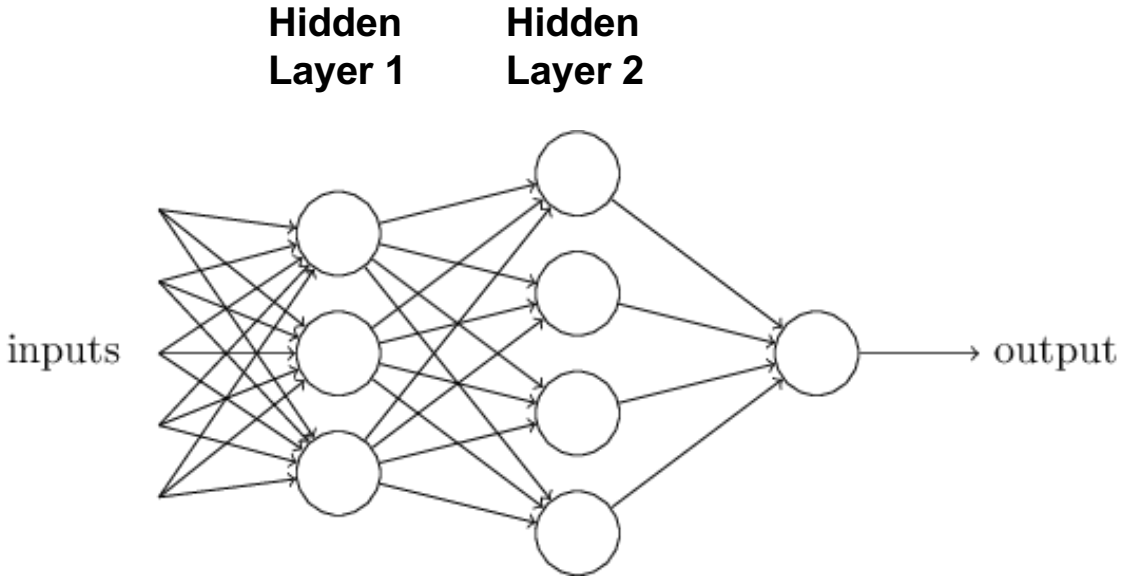

Image source: https://medium.com/@shrutijadon10104776/survey-on-activation-functions-for-deep-learning-9689331ba092

# Neural Networks - multiclass

- Add more perceptrons



# Multi-layer perceptron (MLP)



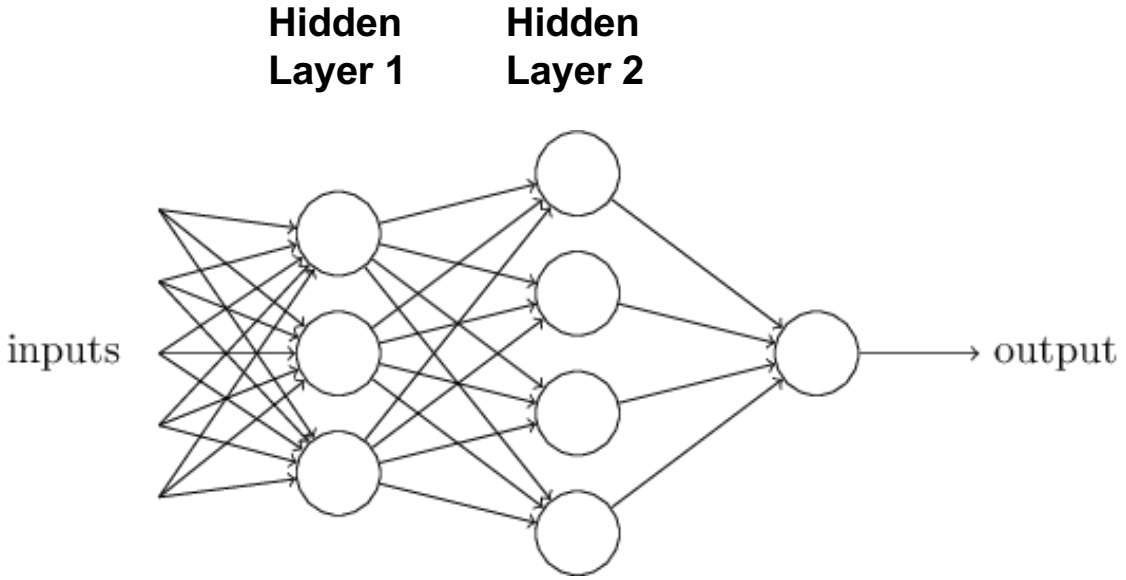

Sets of layers and the connections (weights) between them define the ***network architecture***.

Each layer receives its inputs from the previous layer and **forwards** its outputs to the next layer

Explanation in 3Blue1Brown about the multi-layer perceptron
https://www.youtube.com/watch?v=aircAruvnKk

adapted from Nielsen

# Multi-layer perceptron (MLP)

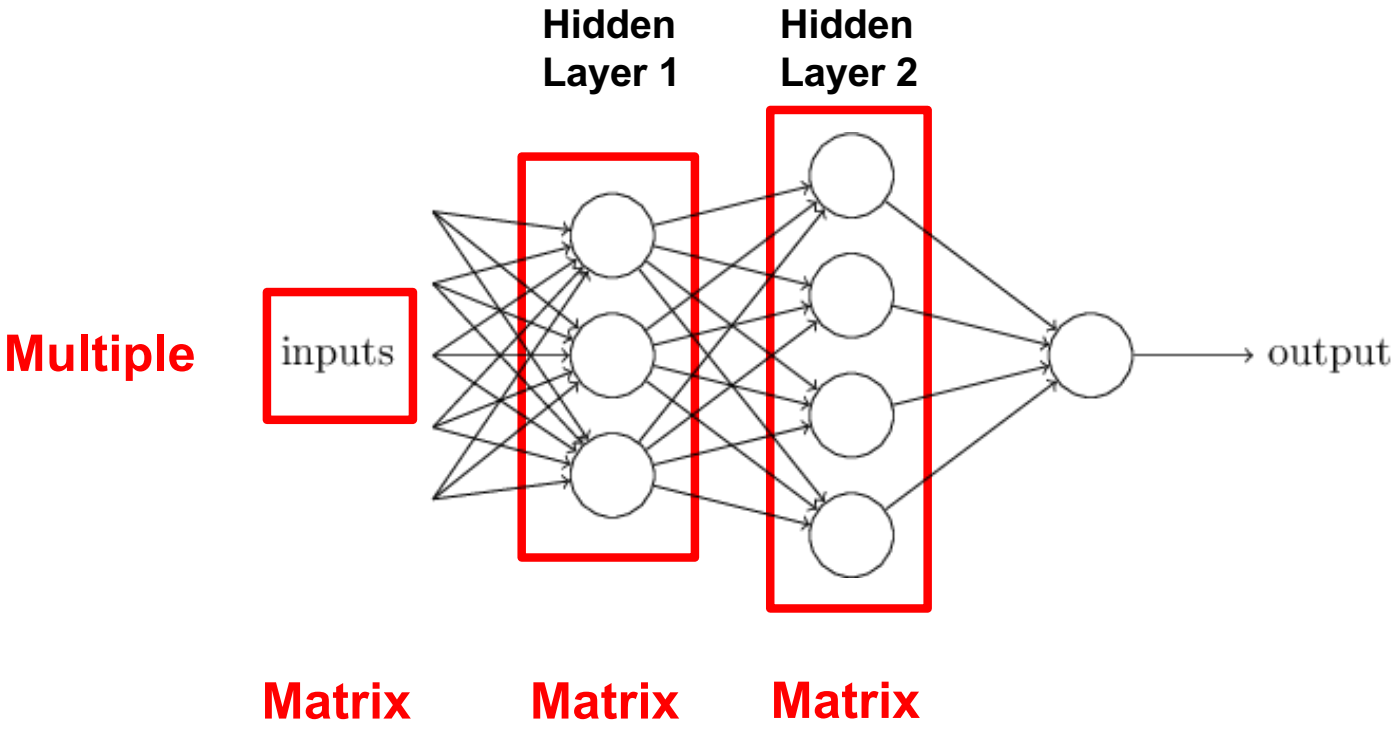



To handle more **complex problems** (than linearly separable ones) we need **multiple layers** → combination of linear boundaries which allow the separation of complex data

Theoretically, using at least two layers (one hidden + one output), with enough hidden units, we can **approximate any function**

adapted from Nielsen

# Multi-layer perceptron (MLP)

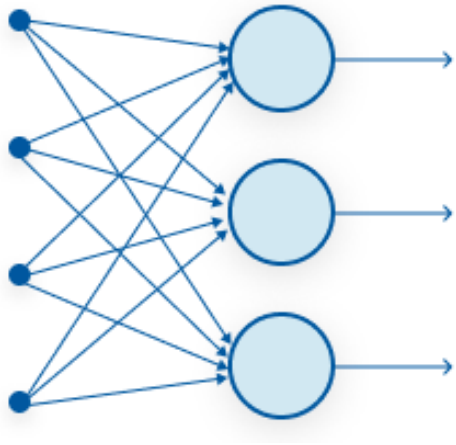



It's all just matrix multiplication!

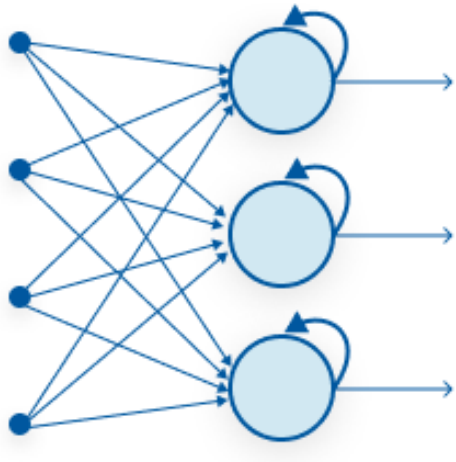
*GPUs -> special hardware for fast/large matrix multiplication.*

# Network structure

- Feed-forward network
  - Directed **acyclic** graph
  - No internal state
  - Simply computes outputs from inputs



- Recurrent network
  - Directed **cyclic** graph
  - Dynamical system with internal states
  - Can memorize information

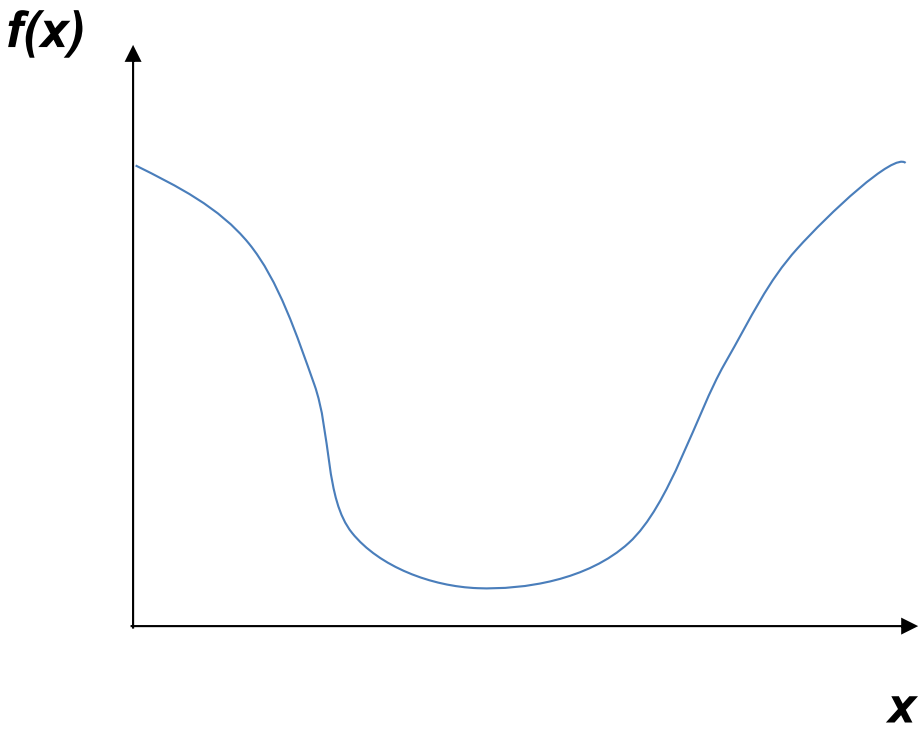


Adapted from Pascal Poupart

Learning the weight matrices $\boldsymbol{W}$

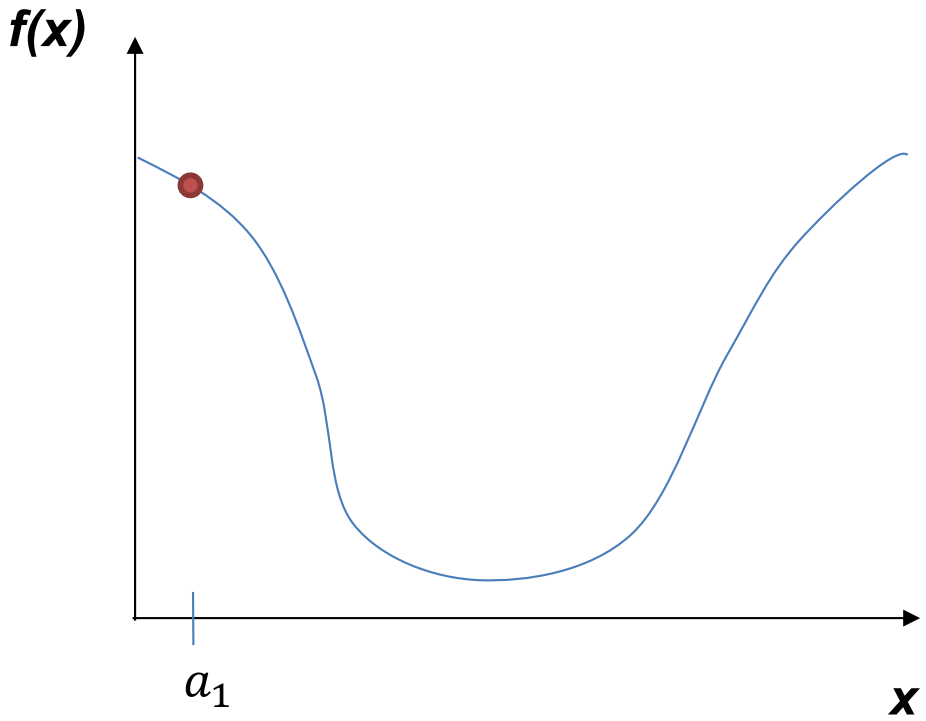
# TRAINING NEURAL NETWORKS

# Gradient descent



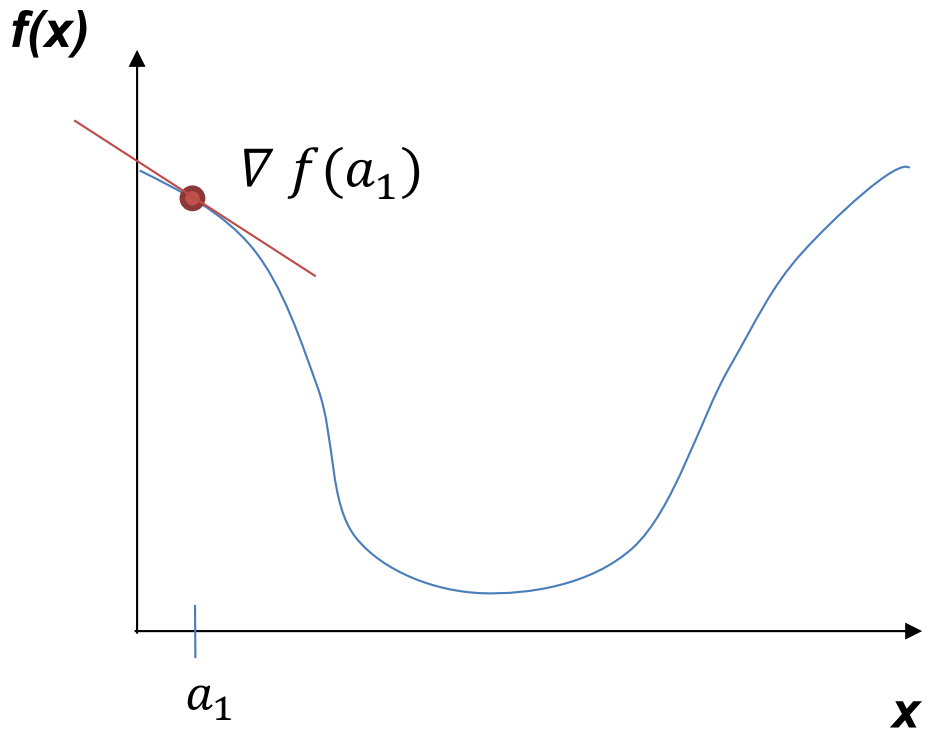
# General approach

Pick random starting point.



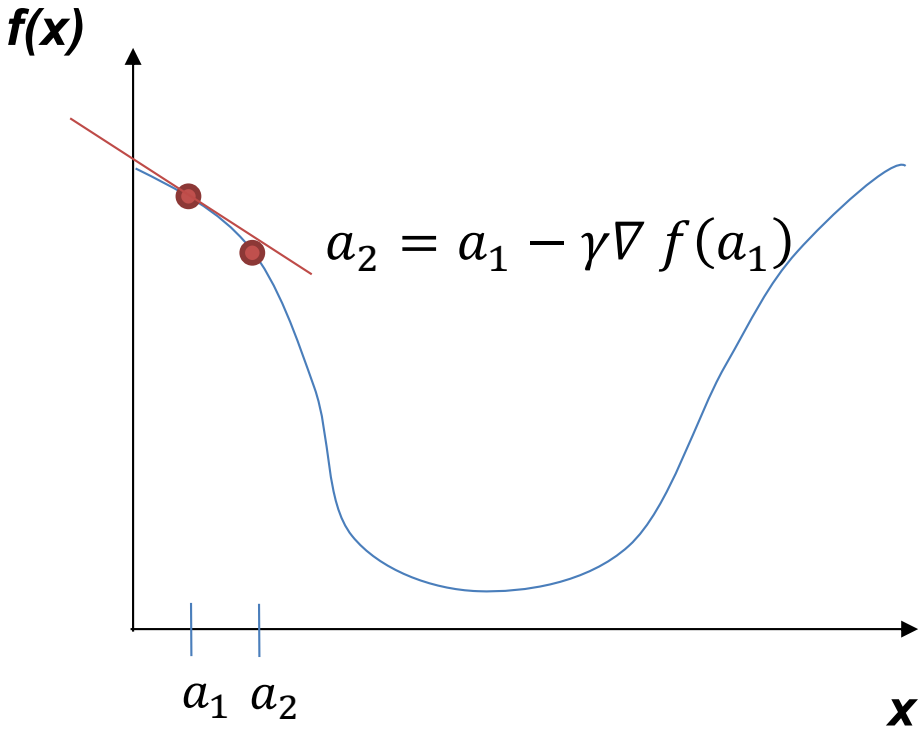
# General approach

Compute gradient at point (analytically or by finite differences)



# General approach

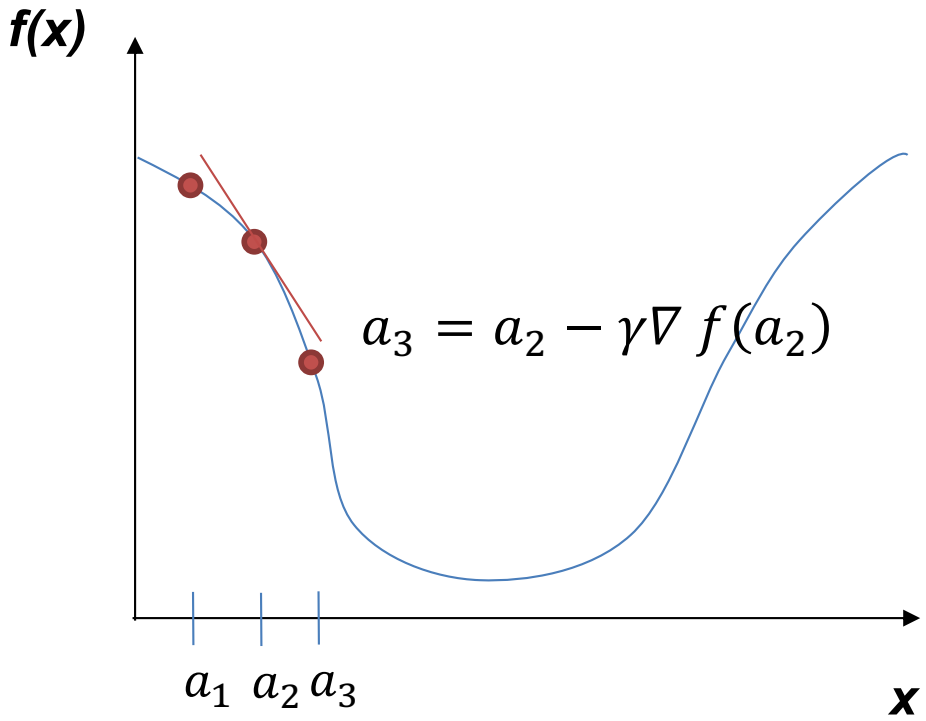
Move along parameter space in direction of negative gradient




$\gamma$ = amount to move
= *learning rate*

# General approach

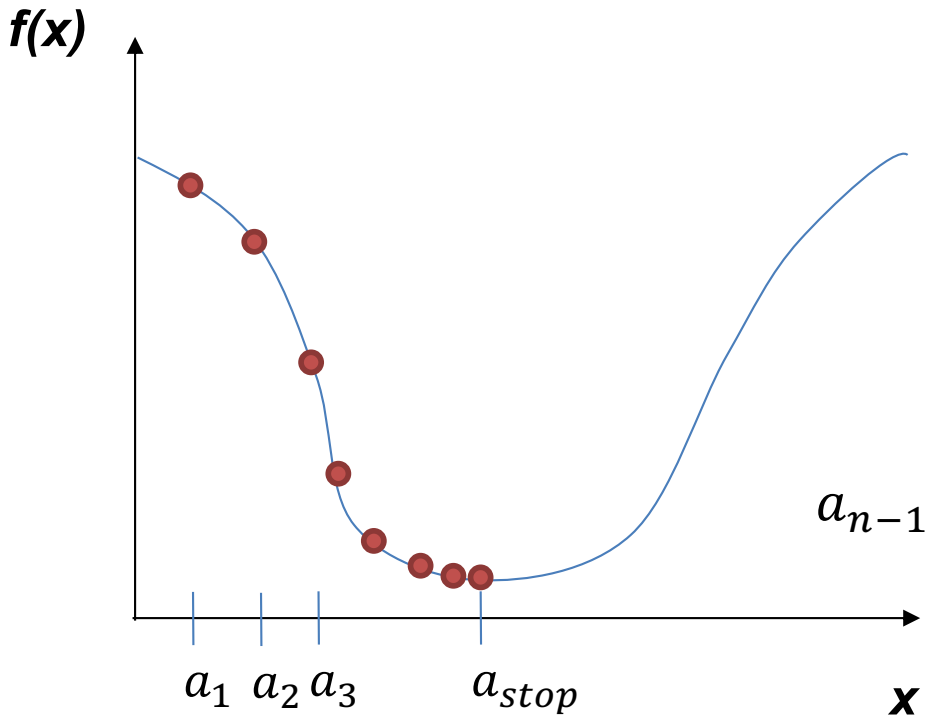
Move along parameter space in direction of negative gradient.




**$\gamma$ = amount to move**
**= *learning rate***

# General approach

Stop when we don't move any more.

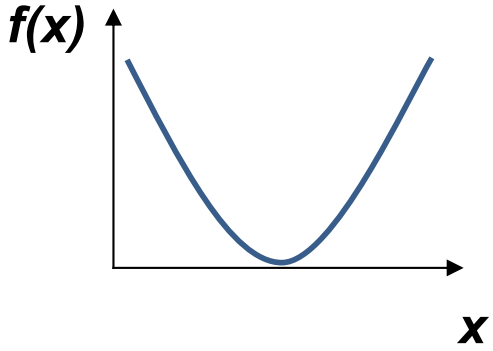



$a_{stop}$:

$$a_{n-1} - \gamma \nabla f(a_{n-1}) = \mathbf{0}$$

# Gradient descent

- Optimizer for functions
- Guaranteed to find optimum for convex functions.
  - Non-convex = find *local* optimum.
  - Most vision problems aren't convex.




- Works for multi-variate functions.
  - Need to compute matrix of *partial derivatives ("Jacobian")*

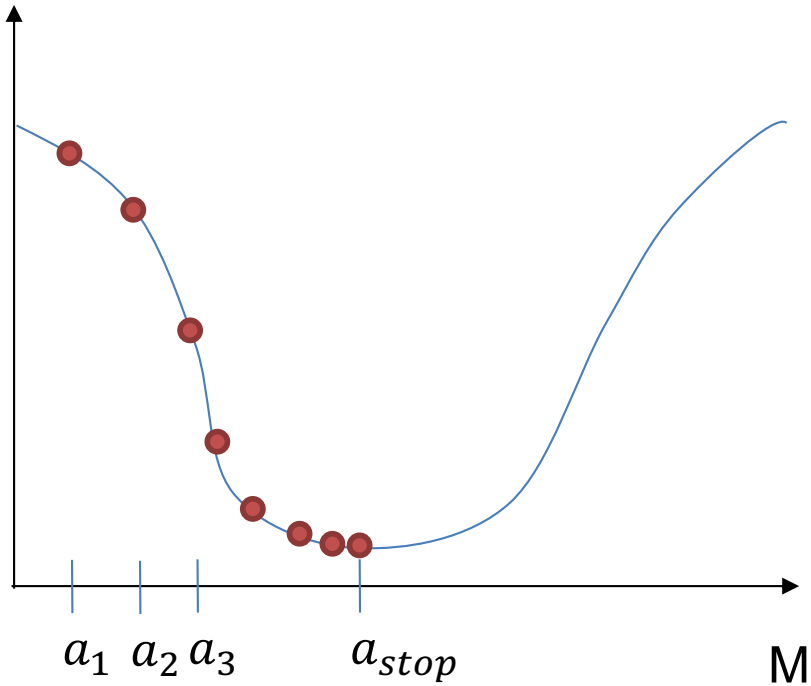
# Train NN with Gradient Descent

- $x^i, y^i$ = *n* training examples
- $f(\boldsymbol{x})$ = feed forward neural network
- L(**x**, y; **θ**) = some *loss function*

- *Loss function* (or cost function) measures how 'good' our network is at classifying the training examples wrt. the parameters of the model, i.e. the perceptron weights.

# Train NN with Gradient Descent

Loss function (Evaluate NN on training data)




Model parameters (perceptron weights)

# Sequential gradient descent

$$w_{ji} \leftarrow w_{ji} - \eta \frac{\partial L}{\partial w_{ji}}$$

$\frac{\partial L}{\partial w_{ji}}$ → Error or Loss

$\eta$ → Learning rate or step length

- In practice, the training is typically done using **sequential gradient descent**, i.e. in each iteration (step), calculate the error and update the weights
- A complete pass over the training set is called an epoch

- How can we compute the gradient efficiently given an arbitrary network structure?
- Answer: backpropagation algorithm
  - propagating errors backward into the network

# Sequential gradient descent

- If each iteration is done:
  - with every input, we have stochastic descent → **on-line training**
  - after accumulating error of all input samples → **batch learning**
  - as a mix of both → **mini-batch learning**
    - This is the usual setup, especially considering the hardware limitations when training models with large datasets

- Important: training with with some samples at each step is not the same as with all the samples (batch) → error surface changes

Explanation in 3Blue1Brown about the training process
https://www.youtube.com/watch?v=IHZwWFHWa-w

# What is an appropriate loss?

- Define some output threshold on detection
- Classification: compare training class to output class
- Zero-one loss $L$ (per class)

$y = true\ label$
$\hat{y} = predicted\ label$

$$L(\hat{y}, y) = I(\hat{y} \neq y),$$

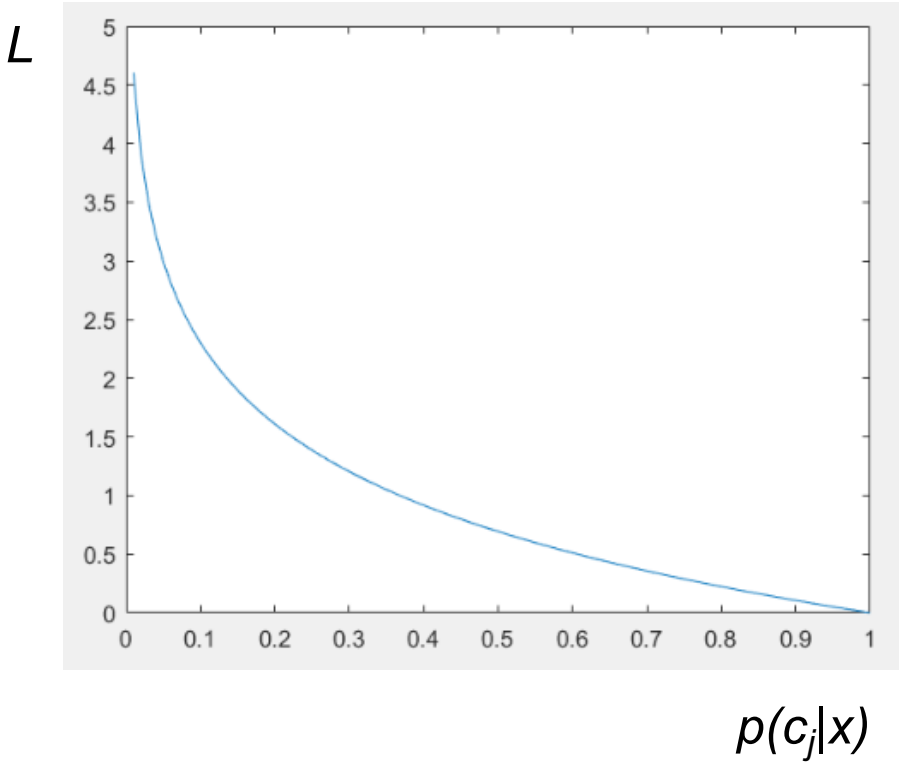
- Is it good?
  - Nope – it's a step function.
  - I need to compute the gradient of the loss.
  - This loss is not differentiable, and 'flips' easily.

# *Cross-entropy loss function*

- Negative log-likelihood

$$L(\boldsymbol{x}, y; \boldsymbol{\theta}) = -\sum_j y_j \log p(c_j|\boldsymbol{x})$$

- Is it a good loss?
  - Differentiable
  - Cost decreases as probability increases



# Typical Loss functions

- Regression
  - Mean Squared Error (MSE) / L2
  - Mean Absolute Error (MAE) / L1
- Binary Classification
  - Binary Cross-Entropy (BCE)
  - Hinge Loss
- Multi-Class Classification
  - Multi-Class Cross-Entropy (CE)

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(\hat{Y}_i - Y_i)^2$$

$$\text{MAE} = \frac{\sum_{i=1}^{n}|y_i - x_i|}{n}$$

$$\text{BCE} = -\frac{1}{m}\sum_{i=1}^{m}(y_i \cdot \log(\hat{y}_i) + (1 - y_i) \cdot \log(1 - \hat{y}_i))$$

$$\text{CE} = -\frac{1}{m}\sum_{i=1}^{m} y_i \cdot \log(\hat{y}_i)$$

- The output of the last layer must be coupled with the loss function:
  - Regression → linear activation
  - Binary classification → sigmoid
  - Multiclass classification → softmax

# Class probability

Special function on last layer - 'S*oftmax*':

- "squashes" a *C*-dimensional vector ***O*** of arbitrary real values to a *C*-dimensional vector σ(***O***) of real values in the range (0, 1) that add up to 1.
- Turns the output into a probability distribution on classes.

$$p(c_k = 1 | \boldsymbol{x}) = \frac{e^{o_k}}{\sum_{j=1}^{C} e^{o_j}}$$

For binary classification, with one output unit, the sigmoid activation function already outputs a probability distribution.

adapted from James Tompkin

# Backpropagation algorithm

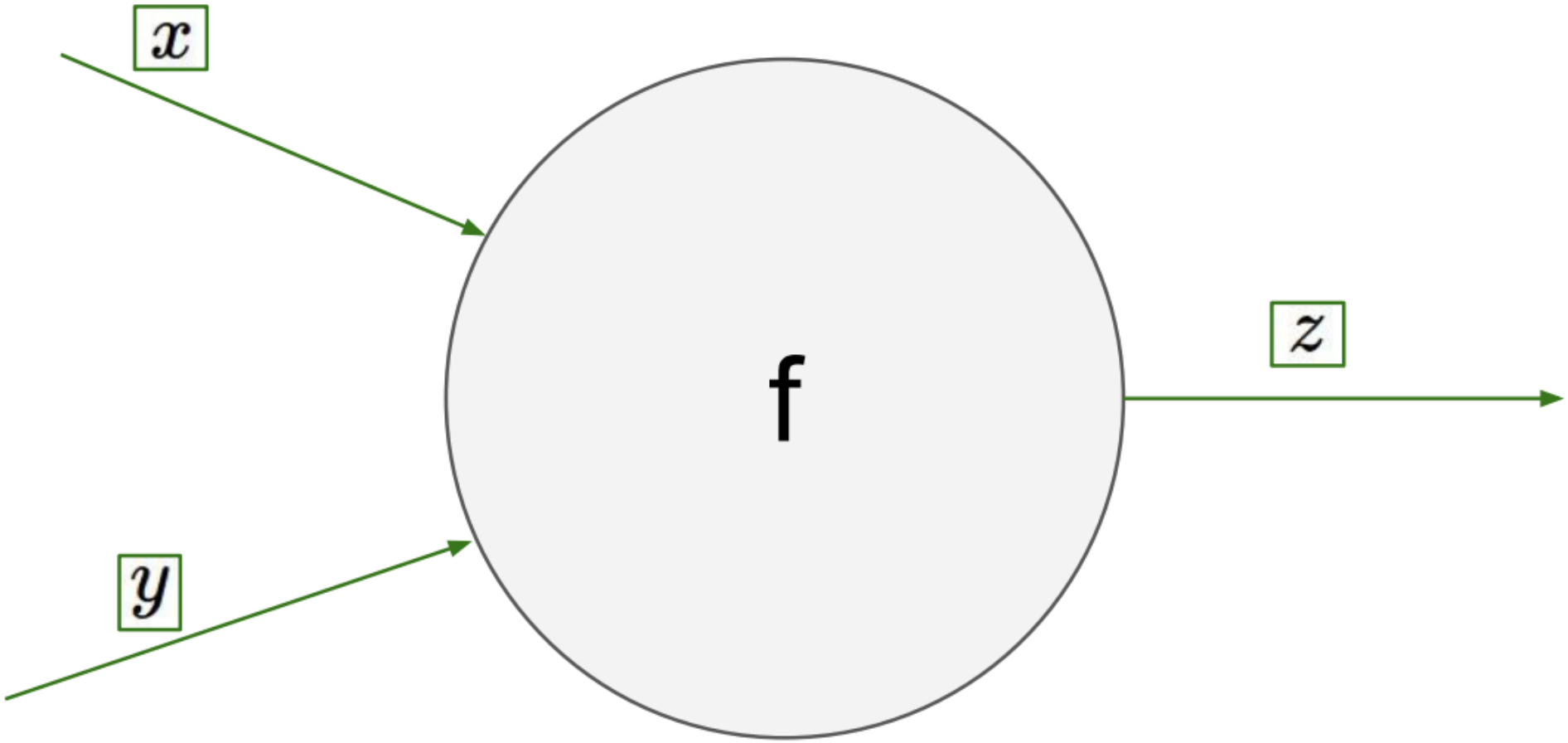
- Two phases:
  - Forward phase: compute output $z_j$, of each unit $j$

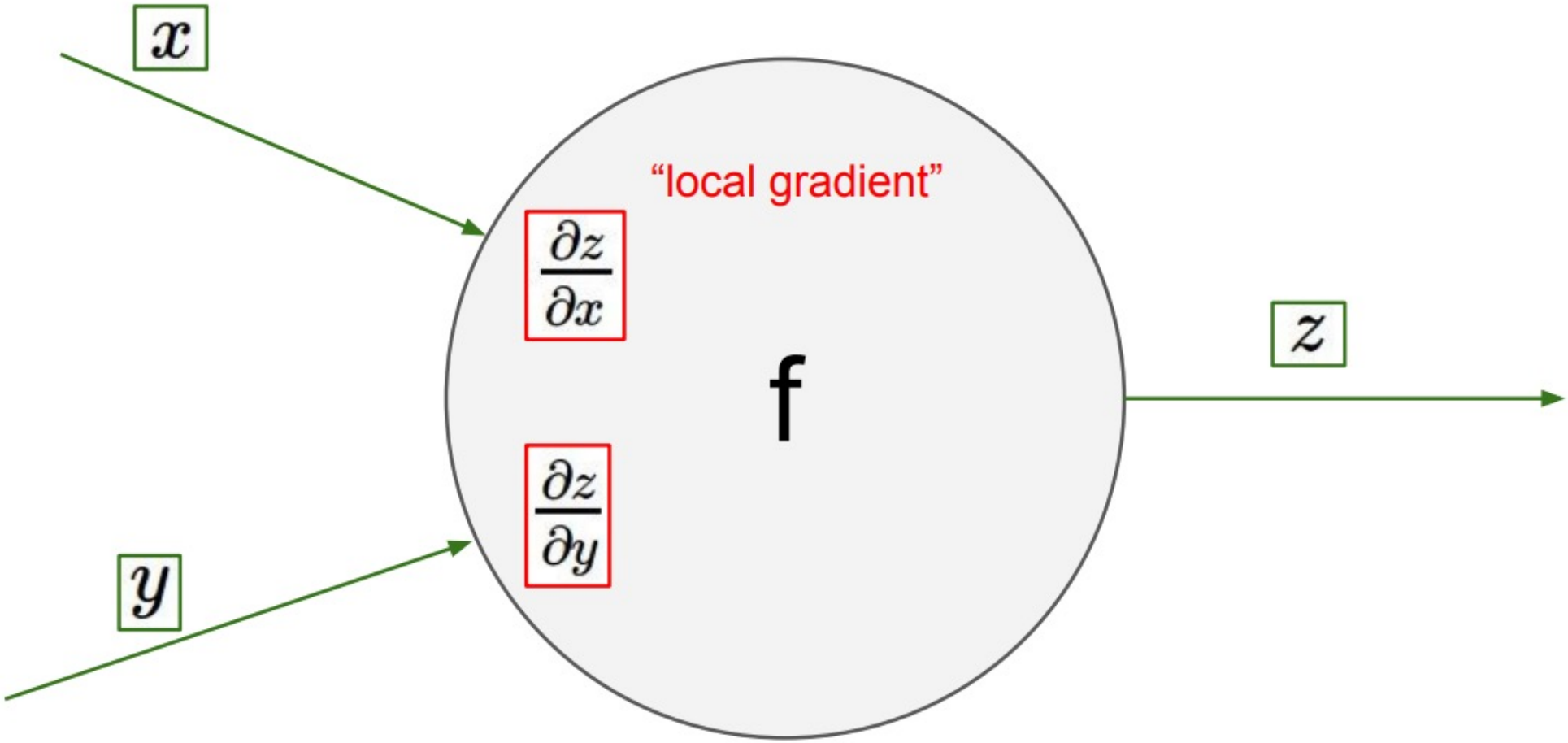
$$z_j = h(a_j) \; where \; a_j = \sum_i w_{ji} z_i$$

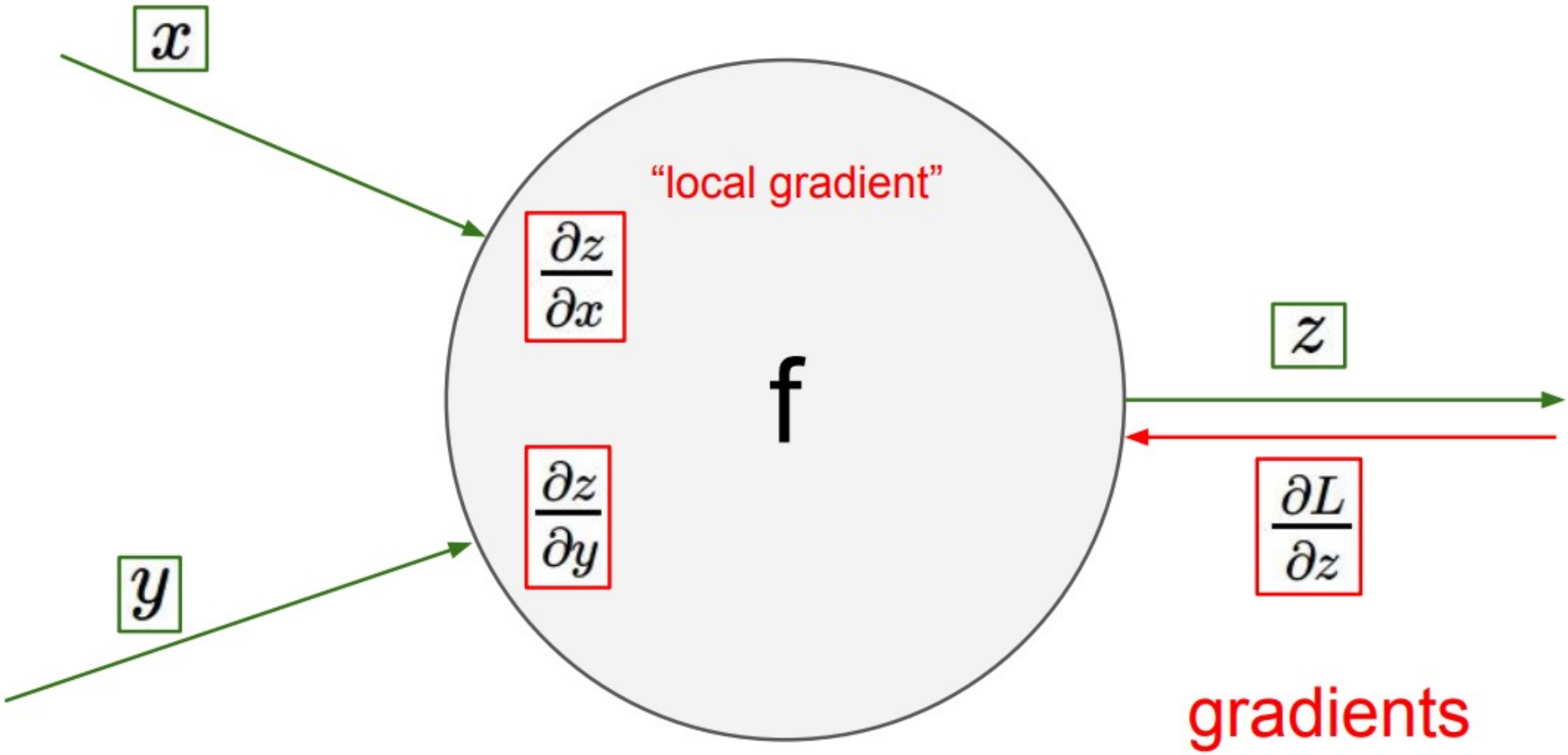
  - Backward phase: compute $\delta_j$, of each unit $j$
    - Use chain rule to recursively compute gradient
      - For each weight $w_{ji}$: $\frac{\partial L}{\partial w_{ji}} = \frac{\partial L}{\partial a_j} \frac{\partial a_j}{\partial w_{ji}} = \delta_j \, z_i$

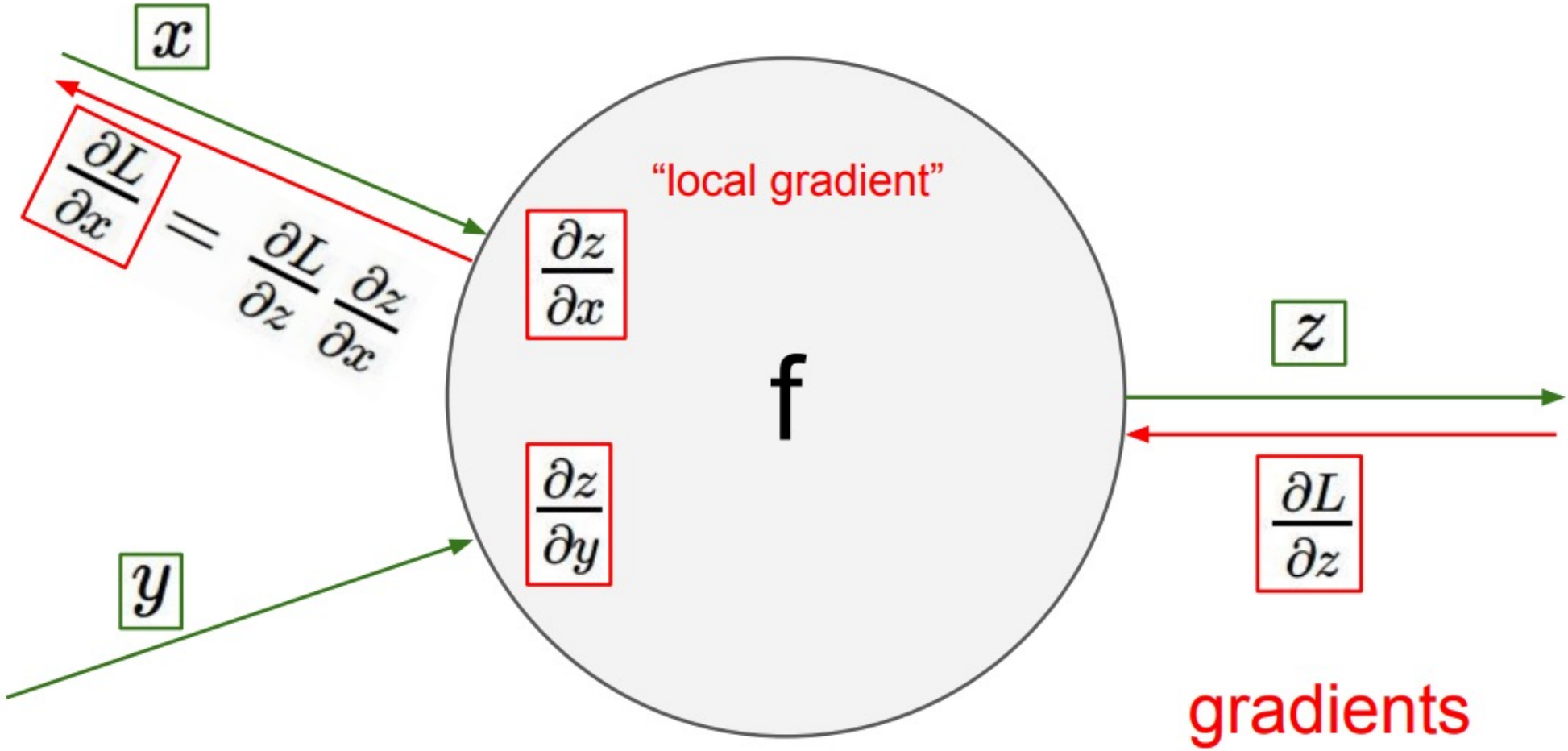
$$\delta_j = \begin{cases} h'(a_j)(z_j - y_j) & \text{base case: } j \text{ is an output unit} \\ h'(a_j) \sum_k w_{kj} \delta_K & \text{recursion: } j \text{ is a hidden unit} \end{cases}$$

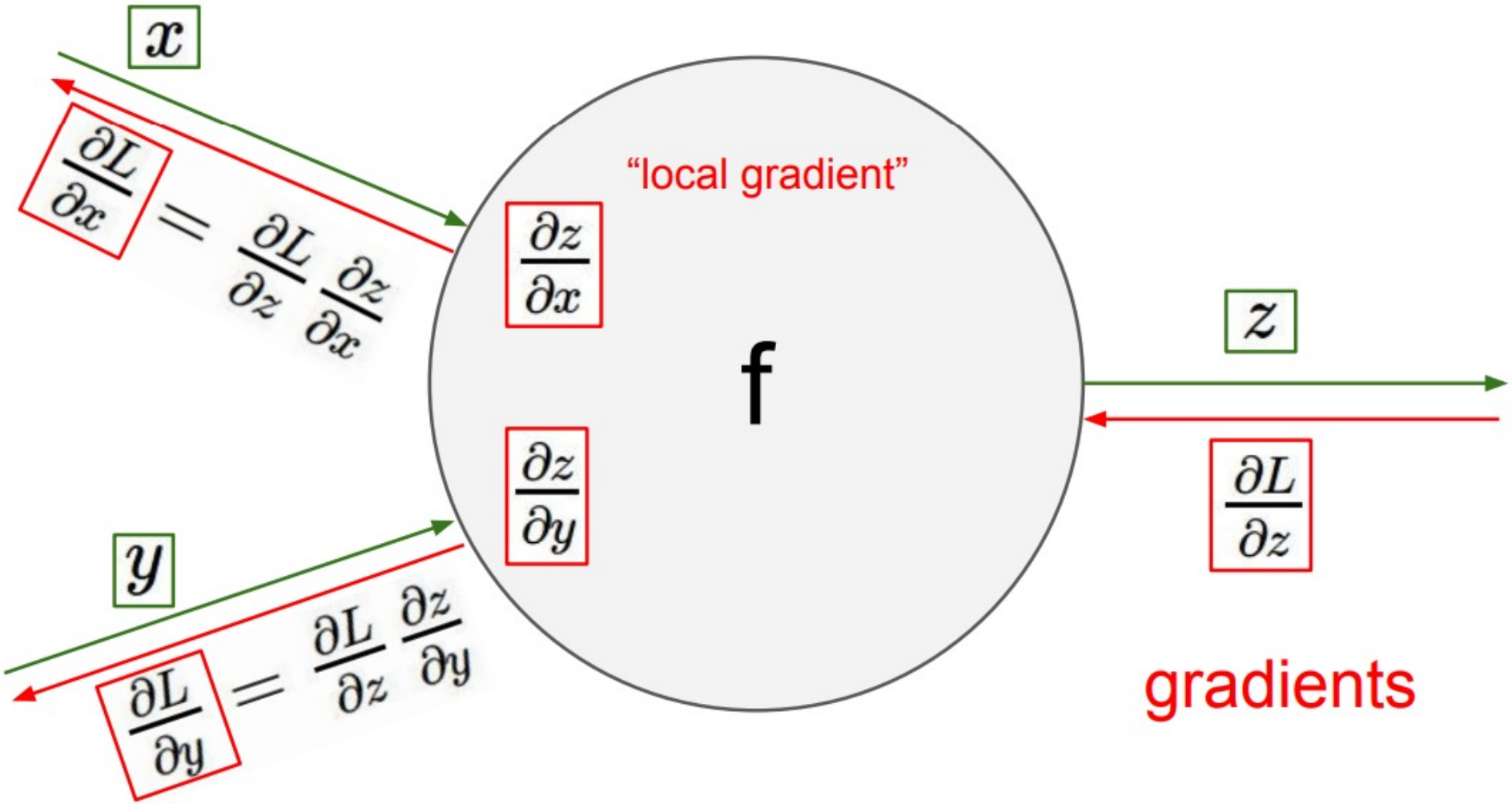
Explanation in 3Blue1Brown about the backpropagation algorithm
https://www.youtube.com/watch?v=Ilg3gGewQ5U


$x$
$y$
f
$z$


$x$
$y$
"local gradient"
$\frac{\partial z}{\partial x}$
$\frac{\partial z}{\partial y}$
f
$z$


$x$
$y$
"local gradient"
$\frac{\partial z}{\partial x}$
$\frac{\partial z}{\partial y}$
f
$z$
$\frac{\partial L}{\partial z}$
gradients

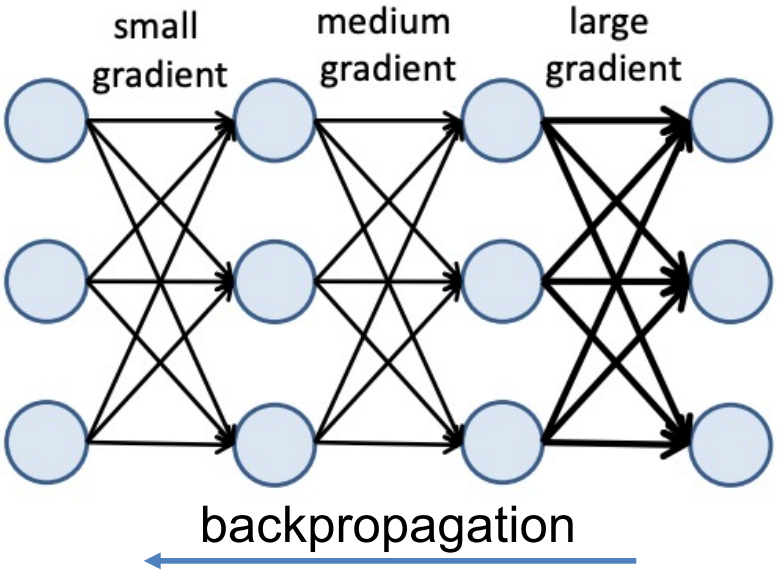

$x$
$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial x}$
"local gradient"
$\frac{\partial z}{\partial x}$
f
$\frac{\partial z}{\partial y}$
$y$
$z$
$\frac{\partial L}{\partial z}$
gradients




Deep learning frameworks use automatic differentiation

# Training Deep NNs

- Deep neural networks of sigmoid and hyperbolic units often suffer from **vanishing gradients**




- ReLU and other training strategies helped overcome this
- In fact, both the forward and backward passes have *L* matrices multiplications → can lead to numerical issues

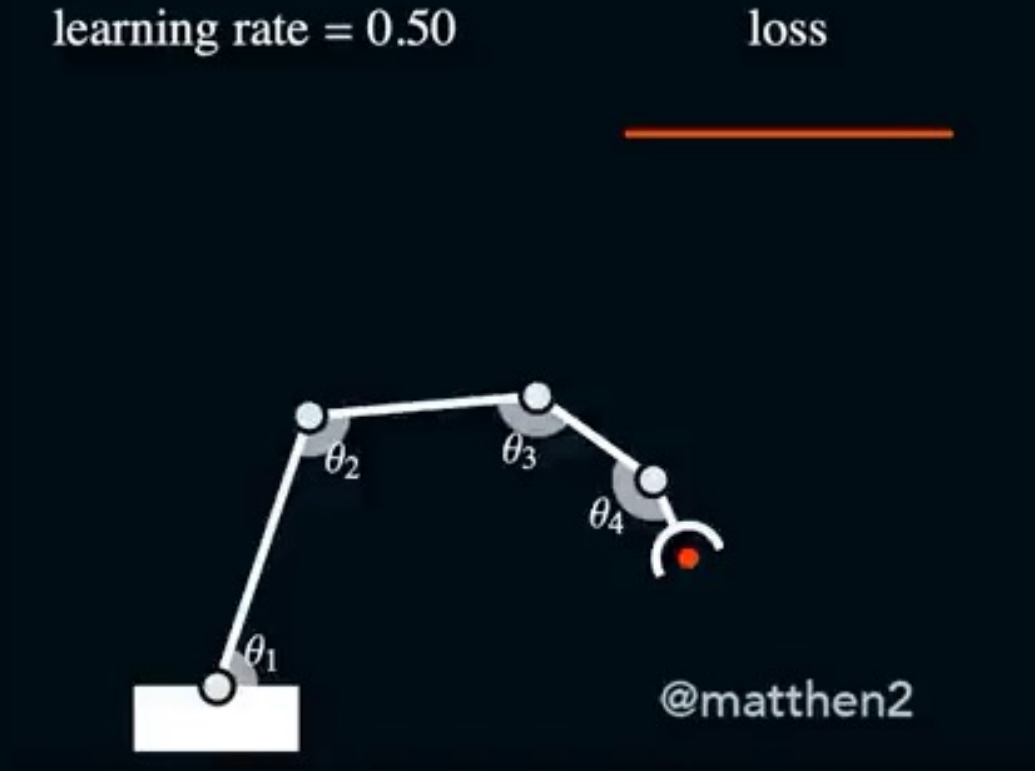
# Vanishing and exploding gradients

- Diagnosis:
  - Exploding: model not learning; cost/loss oscillating or NaN, weights growing exponentially to NaN
  - Vanishing: learning very slow or even stopped very early; weights on the last layers change but those closer to the input don't
- Popular solutions:
  - Limiting the size of the gradients (gradient clipping)
  - Pre-training or proper weight initialization (e.g. Xavier, Kaiming for ReLU layers)
  - Skip connections
  - Batch normalization

Adapted from José Luis Alba

# Learning rate

- LR is the most important hyperparameter in backpropagation and the most **difficult** to set
- Affects: speed of learning, stabilization, escaping from local minima, etc.
  - Too small → slow training, difficult to escape from small gradient areas
  - Too large → oscillating too much (or not converging), not reaching narrow minimums
- Learning rate strategies:
  - Initial large LR to adapt quickly to the initial random weights
  - Modify the LR during training (many possible strategies)



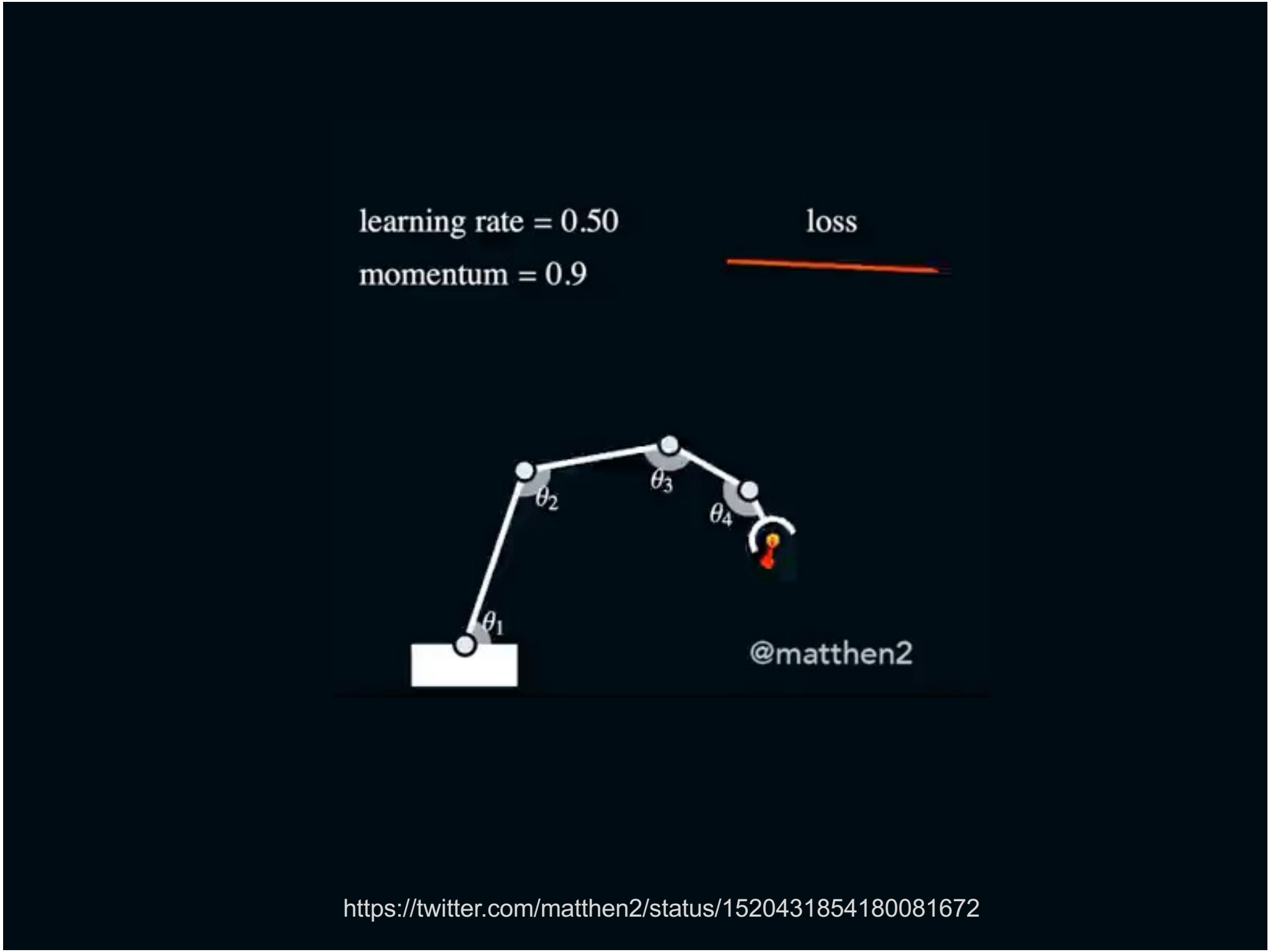

minimize the distance to the target as a function of the angles $\theta_i$

https://twitter.com/matthen2/status/1520427516997025792

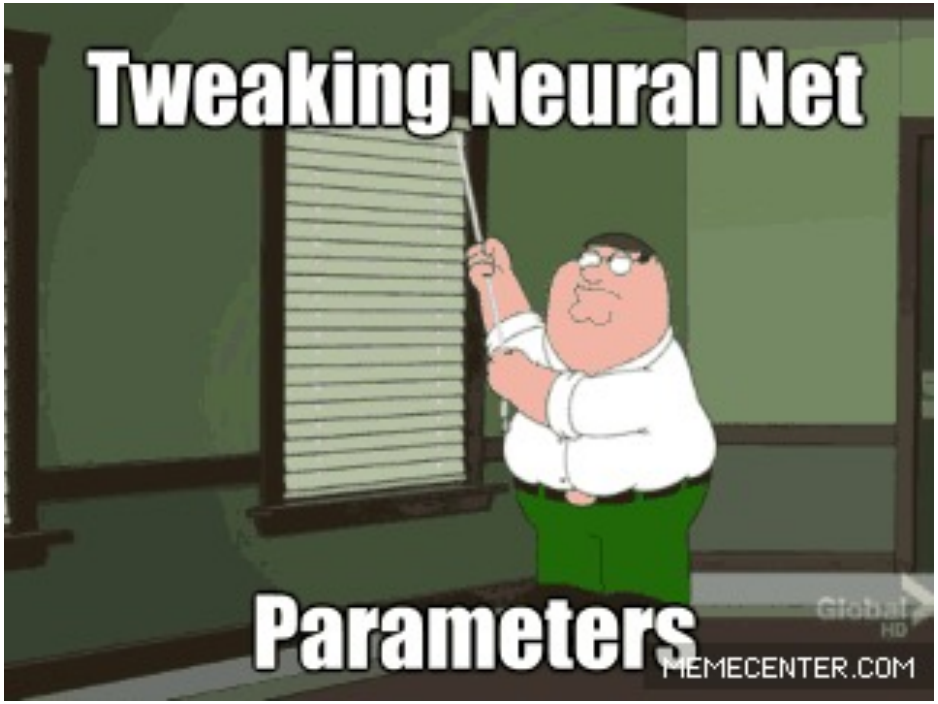
# Faster convergence

- Adaptive gradients
  - Idea: adjust the learning rate of each dimension separately
  - AdaGrad, RMSprop
- Adaptive moment estimation
  - Idea: also replace gradient by its moving average to induce momentum
  - Adam

adapted from Pascal Poupart




https://twitter.com/matthen2/status/1520431854180081672


Tweaking Neural Net
Parameters
MEMECENTER.COM

# Overfitting

- There is serious risk of overfitting, since typically the number of parameters is much larger than the amount of data

- Some solutions:
  - Early stopping
  - Regularization
    - L1 or L2 penalty term in the Loss (penalizes large weights)
    - Dropout
    - Data augmentation

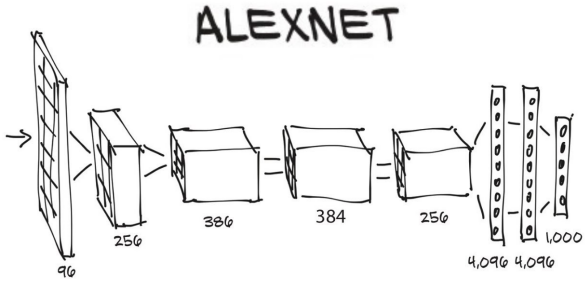
# Overfitting

- Dropout **Perturbations in the network**
  - randomly “drop” some units from the network when training
- Data augmentation **Perturbations in the input data**
  - create new data by applying transformations to the given dataset

# Models

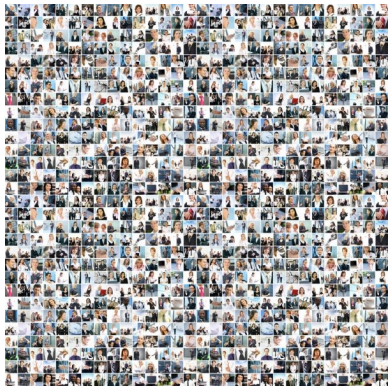
- Unsupervised learning (unlabelled data)
  - Restricted Boltzman Machine (RBM)
  - Autoencoder
- Supervised learning (e.g. classification)
  - Deep Belief Network (DBN)
  - Convolutional Neural Network (CNN)
- Learning in a time series (e.g. forecasting)
  - Recurrent network (e.g. LSTM)
- Learning hierarchical structures
  - Recursive networks

# Deep Learning Revolution




Better Architectures and Learning Strategies

+



A Looooot More Data

+

Processing Power

IMAGENET

www.image-net.org

**22K** categories and **14M** images

- Animals
  - Bird
  - Fish
  - Mammal
  - Invertebrate
- Plants
  - Tree
  - Flower
- Food
- Materials
- Structures
- Artifact
  - Tools
  - Appliances
  - Structures
- Person
- Scenes
  - Indoor
  - Geological Formations
- Sport Activities

Deng, Dong, Socher, Li, Li, & Fei-Fei, 2009

IMAGENET **Large Scale Visual Recognition Challenge**

Steel drum

The Image Classification Challenge:

1,000 object classes

1,431,167 images

**Output:**
Scale
T-shirt
Steel drum
Drumstick
Mud turtle

✔

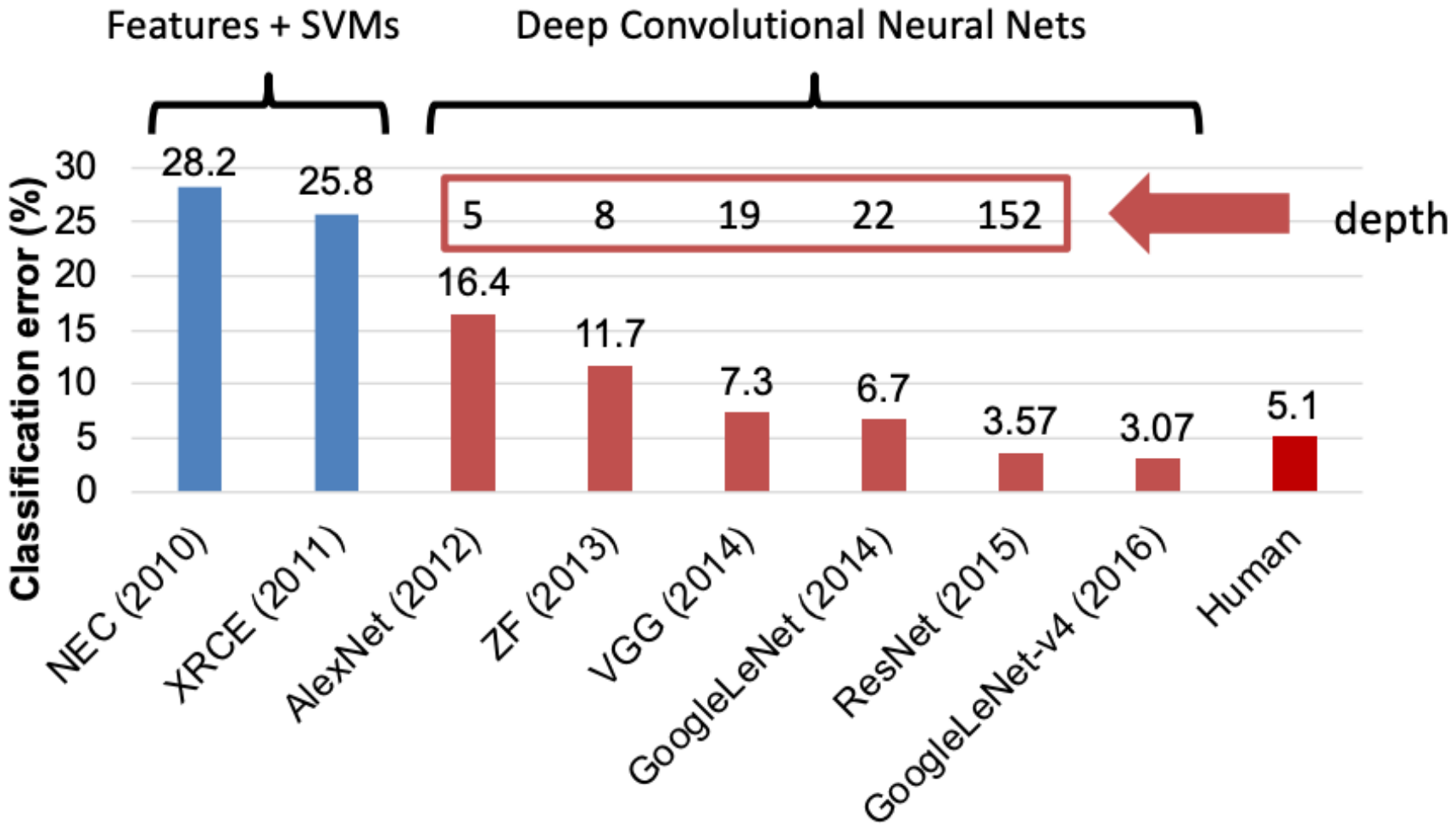
**Output:**
Scale
T-shirt
Giant panda
Drumstick
Mud turtle

✗

Russakovsky et al. arXiv, 2014

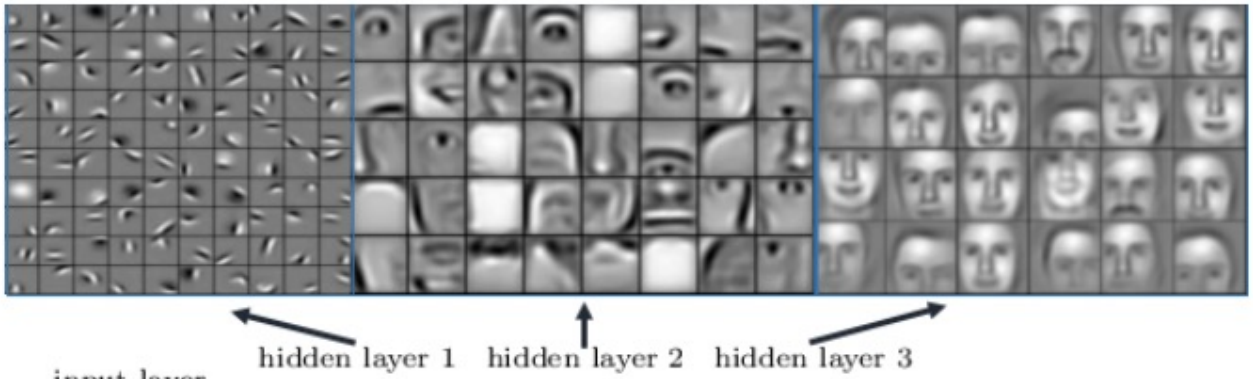
# Image classification

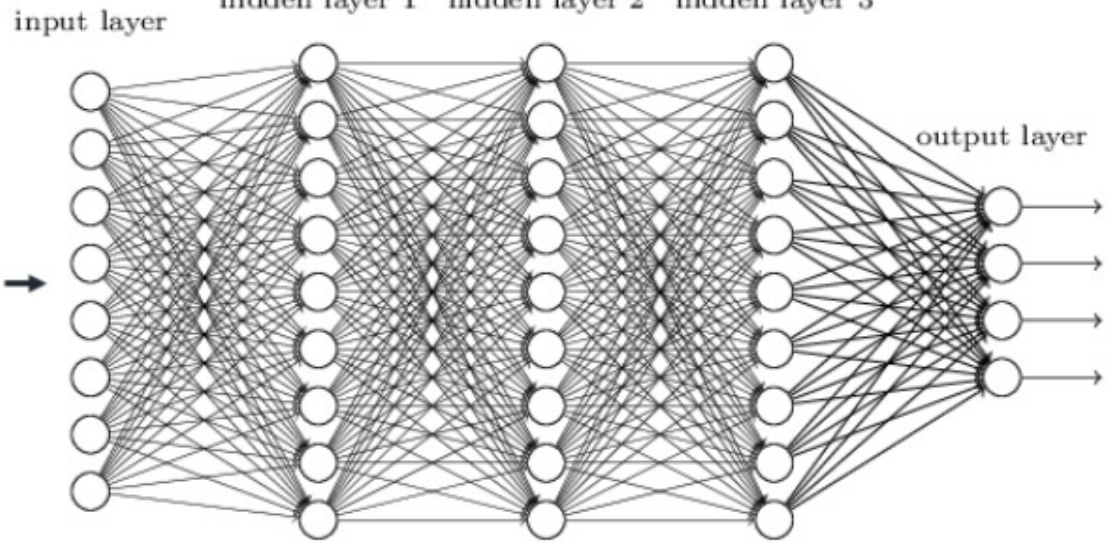
## ImageNet Large Scale Visual Recognition Challenge



# The power of depth (practice)

Deep neural networks learn hierarchical feature representations



# Deep Learning Libraries

- **TensorFlow**: https://www.tensorflow.org/
- **Keras**: http://keras.io
- **PyTorch**: http://pytorch.org
- **MXNet**: https://mxnet.apache.org/

- Architectures easy to create
- CPU/GPU
- Pre-existing models